University of Zagreb
Faculty of Electrical Engineering and Computing

Machine Learning 1 2024/2025

http://www.fer.unizg.hr/en/course/maclea1

Second lab assignment: Linear Discriminative Models and Logistic Regression

Version: 1.1

Deadline: **3 November 2024, 23:59**

Submission rules

By submitting the exercise, you confirm the following points:

1. You did not receive help from another when solving the exercise;
2. You attributed parts of the code that were taken from the Internet by referencing them in comments;
3. You did not use parts of the code from the Internet that are specific to the laboratory exercise;
4. You have not used UI-assistants for coding such as GitHub Copilot (including generative UI tools such as ChatGPT).

Violation of any of the above rules is considered a misdemeanor and results in academic sanctions.

Instructions

The second lab assignment consists of six tasks. Follow the instructions in the text cells below. Solving the lab assignment boils down to **supplementing this notebook**: inserting one or more cells **below** the text of the task, writing the appropriate code, and executing the cells.

Make sure you fully understand the code you've written. When submitting the assignment, you must be able to modify and re-execute your code at the request of the teaching assistant. Furthermore, you need to understand the theoretical basis of what you are doing, within the framework of what we covered in the lecture. Below some tasks you can also find questions that serve as guidelines for a better understanding of the material (**do not write** the answers to the questions in the notebook). Therefore, do not limit yourself only to solving the tasks, but feel free to experiment. This is precisely the purpose of these assignments.

You should do the assignment **independently**. You can consult others on the principle way of solving it, but ultimately you have to do the assignment yourself. Otherwise, the assignment makes no sense.

In [2]:
```
# Load the core libraries...
import sklearn
import matplotlib.pyplot as plt
%pylab inline
```

```
%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib
```

In [3]:
```
def plot_2d_clf_problem(X, y, h=None):
    '''
    Plots a two-dimensional labeled dataset (X,y) and, if function h(x) is given,
    the decision surfaces.
    '''
    assert X.shape[1] == 2, "Dataset is not two-dimensional"
    if h!=None :
        # Create a mesh to plot in
        r = 0.04  # mesh resolution
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, r),
                             np.arange(y_min, y_max, r))
        XX=np.c_[xx.ravel(), yy.ravel()]
        try:
            Z_test = h(XX)
            if Z_test.shape == ():
                # h returns a scalar when applied to a matrix; map explicitly
```

```
                Z = np.array(list(map(h,XX)))
            else :
                Z = Z_test
        except ValueError:
            # can't apply to a matrix; map explicitly
            Z = np.array(list(map(h,XX)))
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        plt.contourf(xx, yy, Z, cmap=plt.cm.Pastel1)

    # Plot the dataset
    plt.scatter(X[:,0],X[:,1], c=y, cmap=plt.cm.tab20b, marker='o', s=50);
```

1. Linear regression as a classifier

In the previous lab exercise, we used a linear regression model for, of course, regression. However, the linear regression model can also be used for **classification**. Although it sounds a bit counterintuitive, it's actually quite simple. Namely, the goal is to learn the function $f(\mathbf{x})$ which predicts the value 1 for positive examples, while it predicts the value 0 for negative examples. In this case, the function $f(\mathbf{x}) = 0.5$ represents the boundary between the classes, i.e., examples for which $h(\mathbf{x}) \geq 0.5$ is valid are classified as positive, while the rest are classified as negative.

Classification using linear regression is implemented in the class `RidgeClassifier`. In the following subtasks, **train** that model on the given data and **display** the obtained boundary between classes. In doing so, turn off regularization ($\alpha = 0$, i.e. `alpha=0`). Also, print the **accuracy** of your classification model (you can use the function `metrics.accuracy_score`). Visualize the data sets using the helper function `plot_clf_problem(X, y, h=None)` given at the beginning of this notebook. `X` and `y` represent the input examples and labels, while `h` represents the model prediction function (e.g., `model.predict`).

In this task, the goal is to consider how the classification model of linear regression behaves on linearly separable and non-separable data.

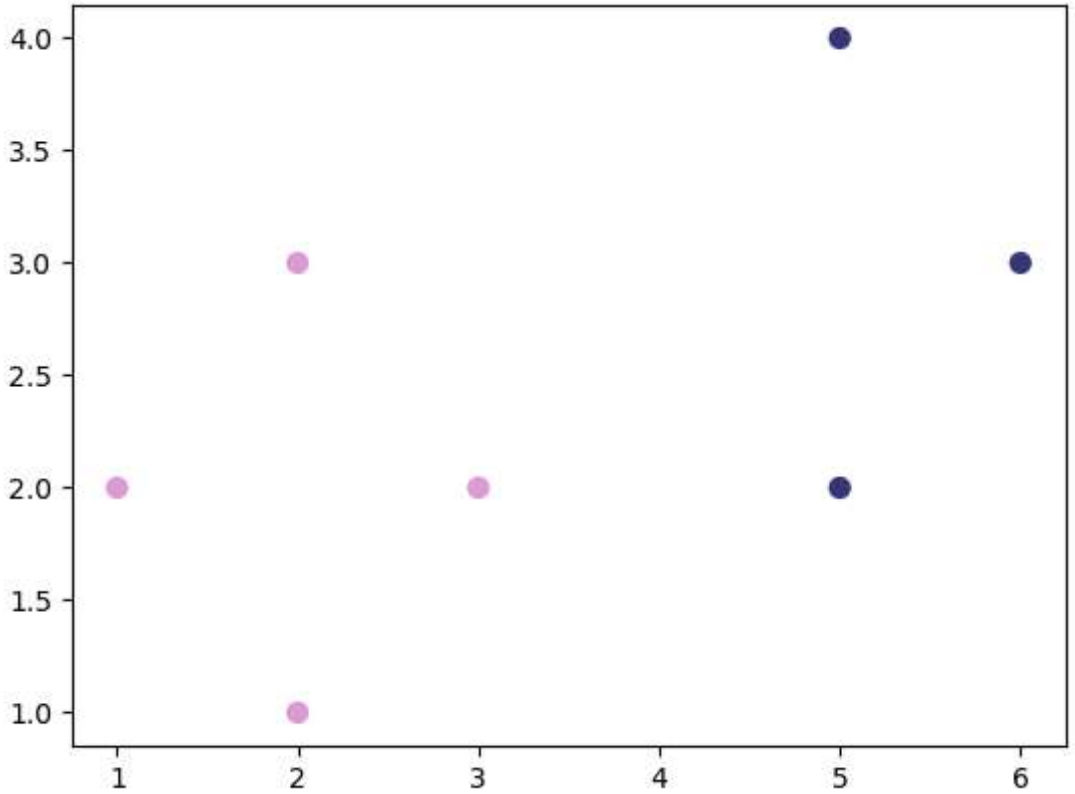
In [6]:
```
from sklearn.linear_model import LinearRegression, RidgeClassifier
from sklearn.metrics import accuracy_score
```

(a)

First, try out the *built-in* model on a linearly separable dataset `seven` ($N = 7$).

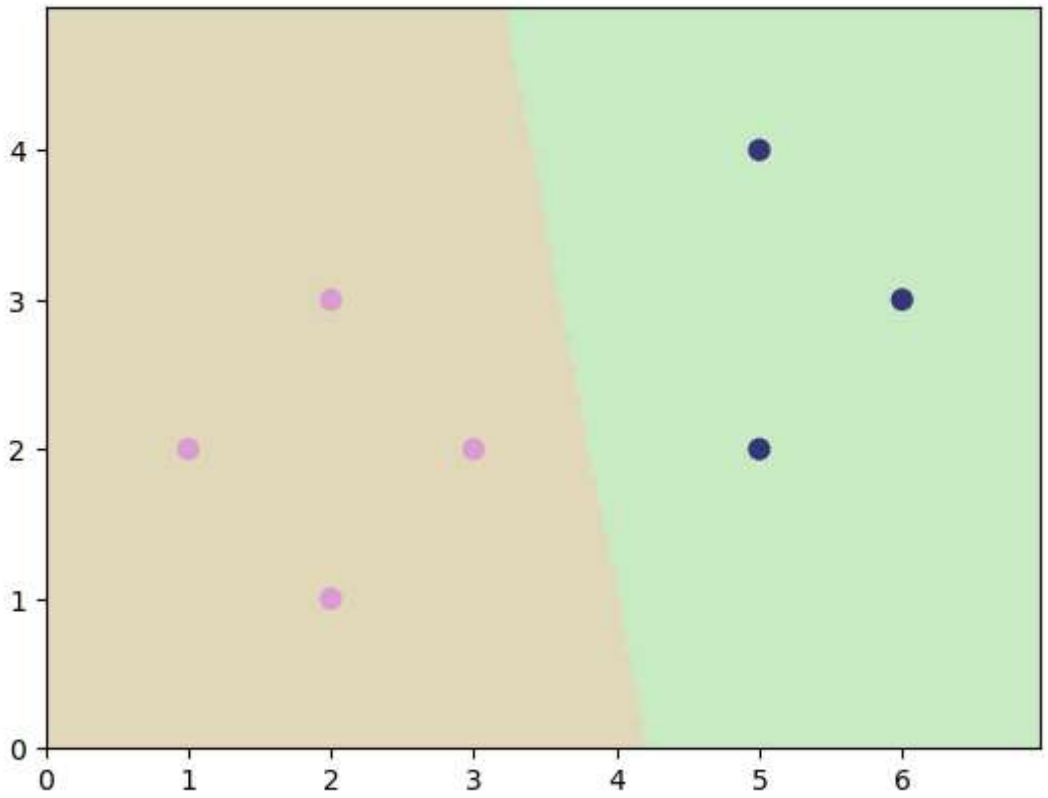
In [8]:
```
seven_X = np.array([[2,1], [2,3], [1,2], [3,2], [5,2], [5,4], [6,3]])
seven_y = np.array([1, 1, 1, 1, 0, 0, 0])
```

In [10]:
```
# Your code here
plot_2d_clf_problem(seven_X, seven_y)
```



To make sure that the implementation you tried is nothing more than simple linear regression, write code that arrives at an equivalent solution using only the `LinearRegression` class. You can define the prediction function, which you pass as the third argument `h` to the `plot_2d_clf_problem` function, with a lambda-expression: `lambda x : model.predict(x) >= 0.5`.

In [14]:
```
# Your code here
model = LinearRegression().fit(seven_X, seven_y)
plot_2d_clf_problem(seven_X, seven_y, lambda x : model.predict(x) >= 0.5)
```



Q: How would the boundary between classes be defined if we used the class labels -1 and 1 instead of 0 and 1?

In [2]:
```
seven_y2 = np.array([-1, -1, -1, -1, 1, 1, 1])
model = LinearRegression().fit(seven_X, seven_y2)
plot_2d_clf_problem(seven_X, seven_y2, lambda x : model.predict(x) >= 0)
```

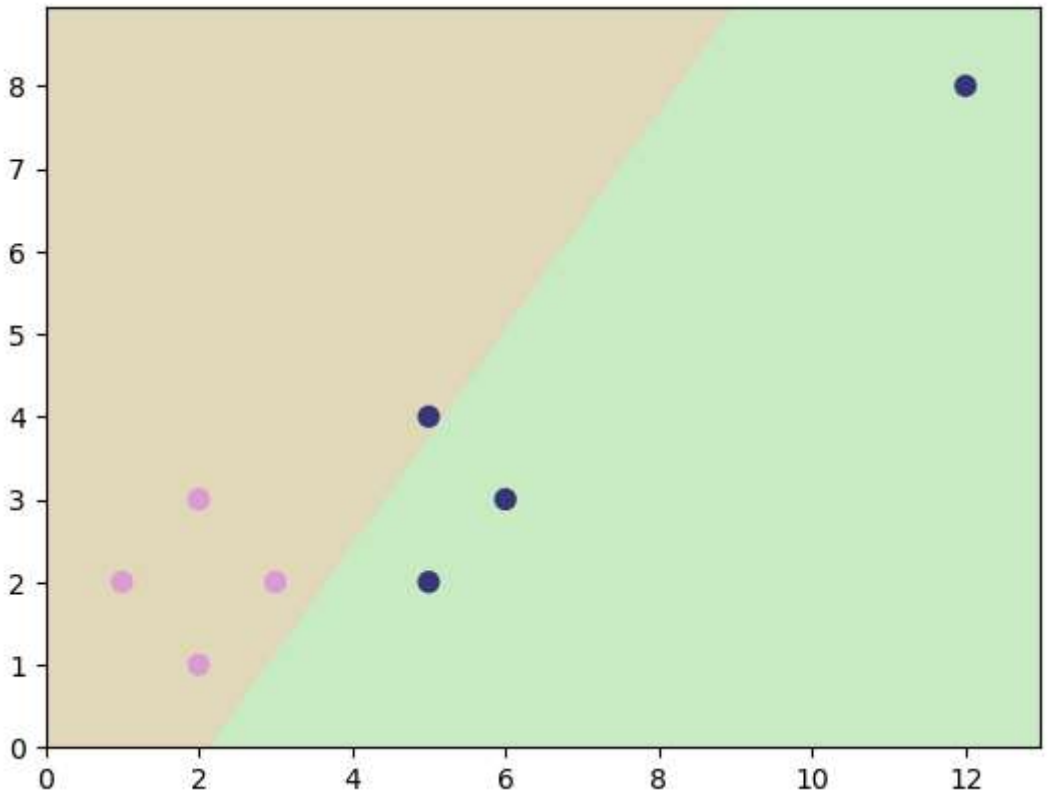
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 seven_y2 = np.array([-1, -1, -1, -1, 1, 1, 1])
      2 model = LinearRegression().fit(seven_X, seven_y2)
      3 plot_2d_clf_problem(seven_X, seven_y2, lambda x : model.predict(x) >= 0)

NameError: name 'np' is not defined
```

(b)

Try the same on the linearly separable data set `outlier` ($N = 8$):

In [18]:
```
outlier_X = np.append(seven_X, [[12,8]], axis=0)
outlier_y = np.append(seven_y, 0)
```

In [30]:
```
# Your code here
model = LinearRegression().fit(outlier_X, outlier_y)
plot_2d_clf_problem(outlier_X, outlier_y, lambda x : model.predict(x) >= 0.5)
```



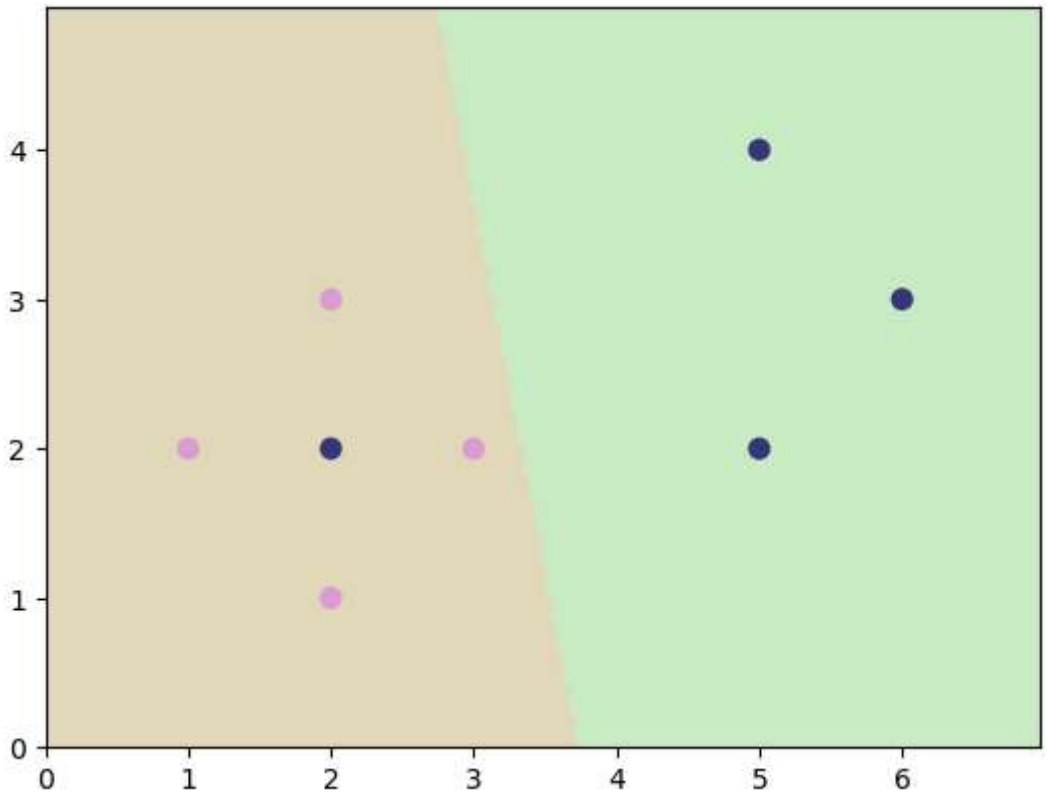
Q: Why doesn't the model achieve complete accuracy even though the data is linearly separable?

(c)

Finally, try the same on the linearly inseparable data set `unsep` ($N = 8$):

In [22]:
```
unsep_X = np.append(seven_X, [[2,2]], axis=0)
unsep_y = np.append(seven_y, 0)
```

In [32]:
```
# Your code here
model = LinearRegression().fit(unsep_X, unsep_y)
plot_2d_clf_problem(unsep_X, unsep_y, lambda x : model.predict(x) >= 0.5)
```



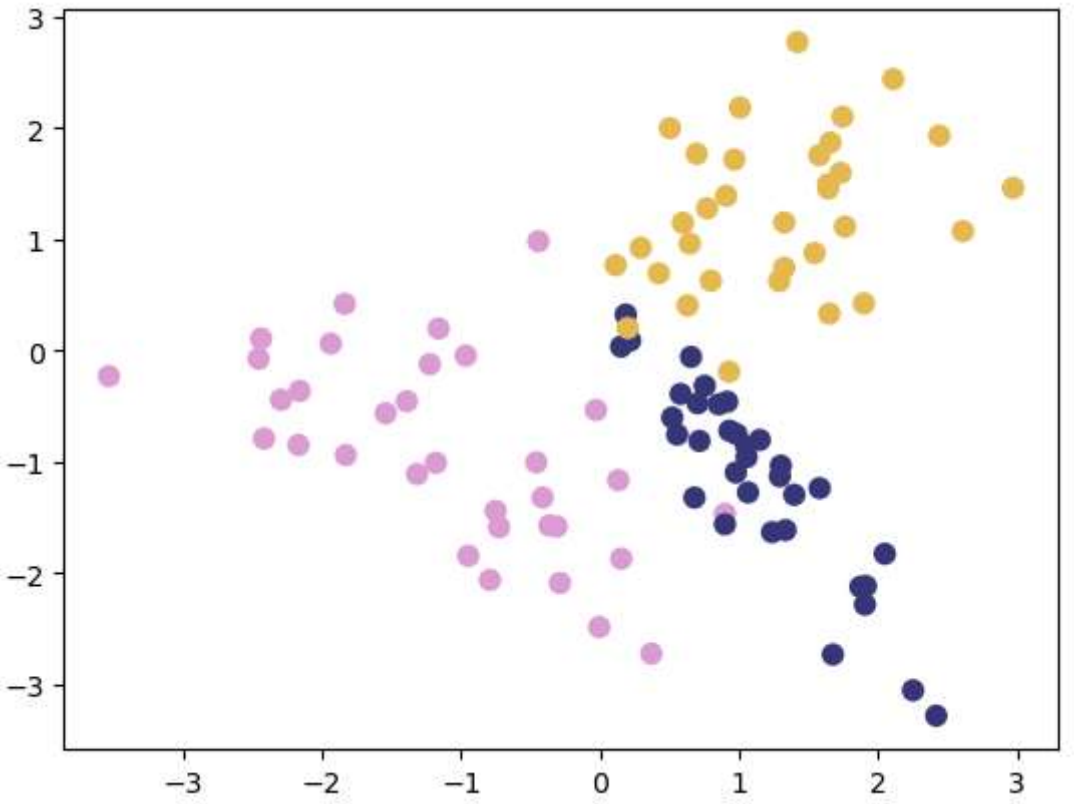
Q: It is obvious why the model is not able to achieve full accuracy on this data set. However, do you think the problem is in the model or in the data? Explain.

2. Mutliclass classification

There are several ways that binary classifiers can be used for multiclass classification. The most commonly used scheme is the so-called **one-vs-rest** (eng. *one-vs-rest*, OVR), in which one classifier h_j is trained for each of the K classes. Each classifier h_j is trained to separate examples of class j from examples of all other classes, and the example is classified into the class j for which $h_j(\mathbf{x})$ is maximal.

Use the `datasets.make_classification` function to generate a three-class random two-dimensional dataset and display it using the `plot_2d_clf_problem` function. For simplicity, assume that there are no redundant features and that each of the classes is "packed" into exactly one group.

In [52]:
```
from sklearn.datasets import make_classification
# Your code here
X, y = make_classification(n_features=2, n_redundant=0, n_classes=3, n_clusters_per_class=1)
plot_2d_clf_problem(X,y)
```

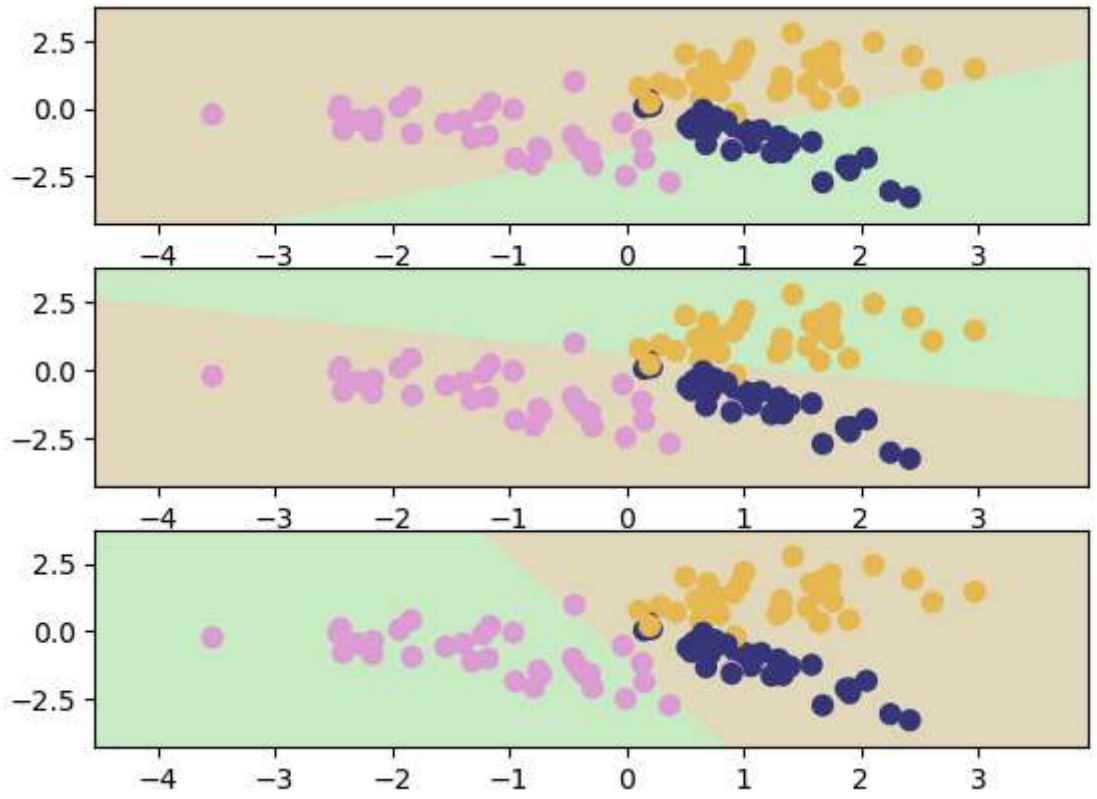


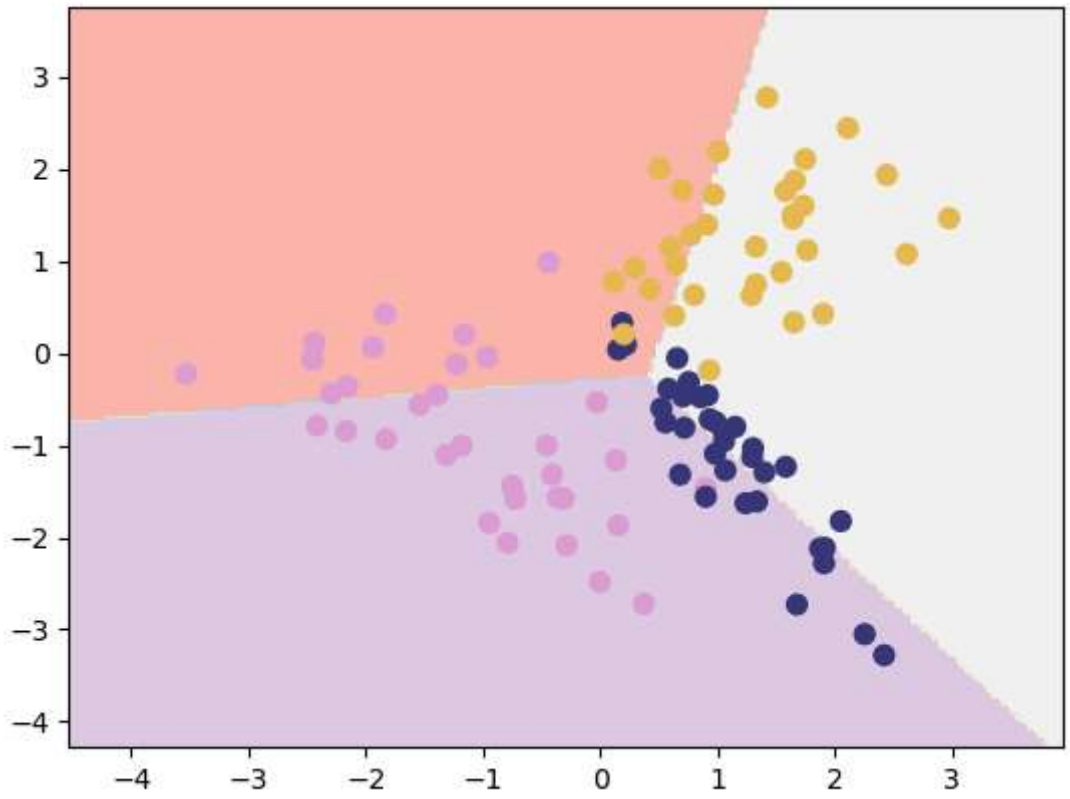
Train three binary classifiers, h_1, h_2, and h_3, and display the boundaries between the classes (three graphs). Then define $h(\mathbf{x}) = \text{argmax}_j h_j(\mathbf{x})$ (write your own `predict` function that does this) and display the class boundaries for that model. Then make sure that you would get an identical result by applying the `RidgeClassifier` model directly, since that model actually internally implements a one-versus-other scheme for a multi-class problem.

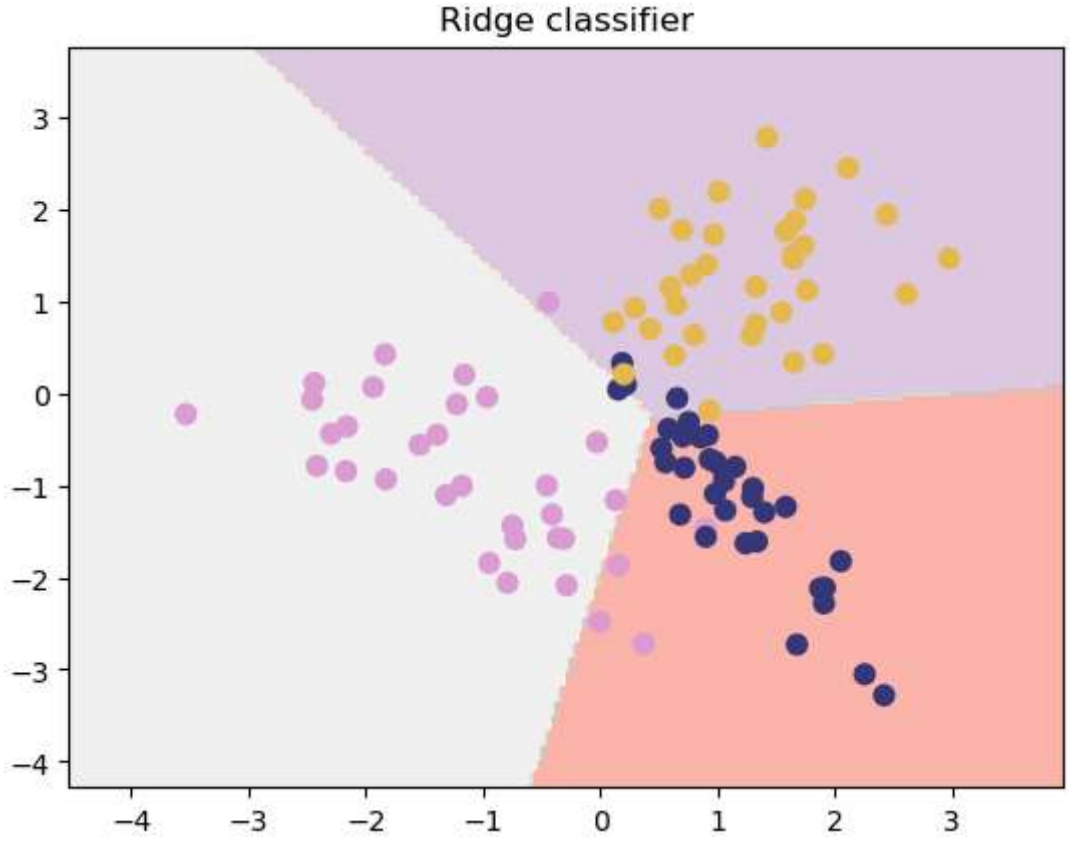
Q: An alternative scheme is the one called **one-vs-one** (engl, *one-vs-one*, OVO). What is the advantage of the OVR scheme over the OVO scheme? And vice versa?

In [117…

```
# Your code here
from sklearn.linear_model import RidgeClassifier

model_1 = LinearRegression().fit(X, np.where(y == 0, 0, 1))
model_2 = LinearRegression().fit(X, np.where(y == 1, 0, 1))
model_3 = LinearRegression().fit(X, np.where(y == 2, 0, 1))
plt.subplot(311)
plot_2d_clf_problem(X, y, lambda x : model_1.predict(x) >= 0.5)
plt.subplot(312)
plot_2d_clf_problem(X, y, lambda x : model_2.predict(x) >= 0.5)
plt.subplot(313)
plot_2d_clf_problem(X, y, lambda x : model_3.predict(x) >= 0.5)

plt.figure()
plot_2d_clf_problem(X,y,
                    lambda x : np.argmax([model_1.predict([x])[0], model_2.predict([x])[0], model_3.predict([x])[0]]))

ridge = RidgeClassifier().fit(X,y)
plt.figure()
plt.title('Ridge classifier')
plot_2d_clf_problem(X,y, lambda x : ridge.predict(x))

#plot_2d_clf_problem(X, y, )
```


2.5
0.0
−2.5
−4
−3
−2
−1
0
1
2
3


3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3



3. Logistic regression

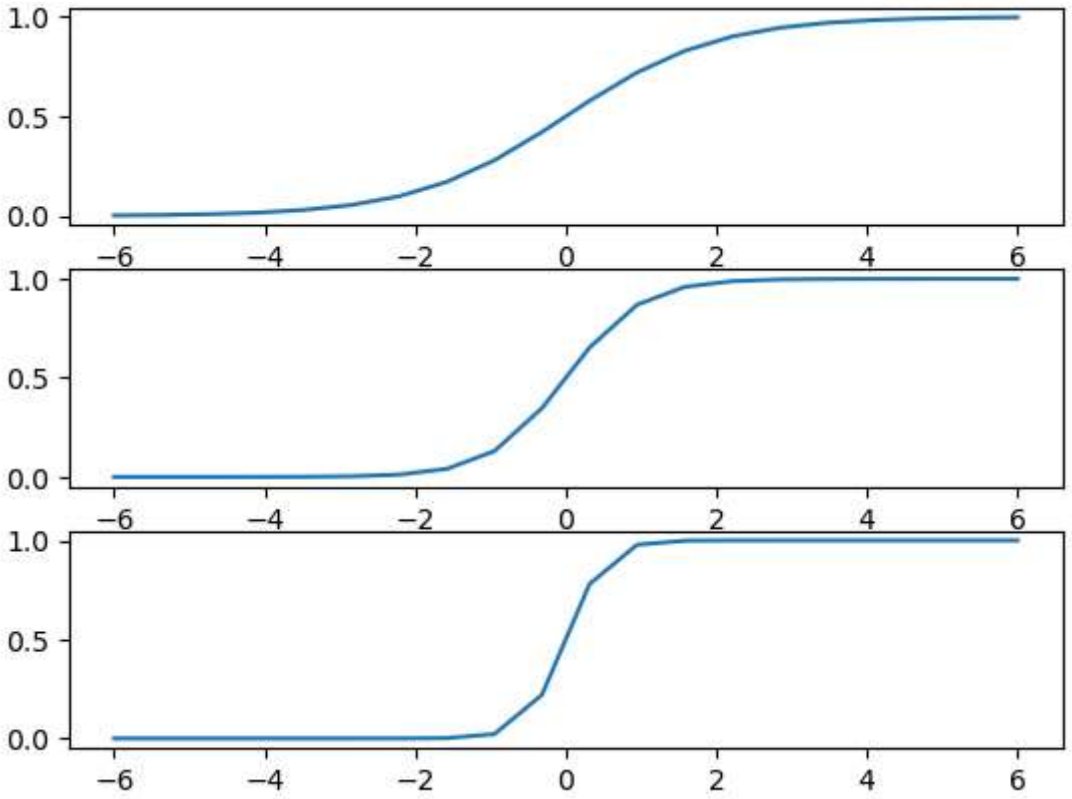
This task deals with a probabilistic discriminative model, **logistic regression**, which, despite its name, is a classification model.

Logistic regression is a typical representative of the so-called **generalized linear models** which are of the form: $h(\mathbf{x}) = f(\mathbf{w}^\mathsf{T}\tilde{\mathbf{x}})$. The logistic function for the function f uses the so-called **logistic** (sigmoidal) function $\sigma(x) = \frac{1}{1+exp(-x)}$.

(a)

Define the logistic (sigmoidal) function $\mathrm{sigm}(x) = \frac{1}{1+\exp(-\alpha x)}$ and show it for $\alpha \in \{1, 2, 4\}$.

In [86]:
```
# Your code here
def sigmoid(x, alpha=1):
    return 1 / (1 + np.exp(-alpha * x))

plt.figure()
z = np.linspace(-6,6, 20)
plt.subplot(311)
plt.plot(z, sigmoid(z,1))
plt.subplot(312)
plt.plot(z, sigmoid(z,2))
plt.subplot(313)
plt.plot(z, sigmoid(z,4))
plt.show()
```



Q: Why is the sigmoid function a suitable choice for the activation function of a generalized linear model?

Q: What influence does the factor α have on the shape of the sigmoid? What does this mean for a logistic regression model (ie, how does the output of the model depend on the norm of the weight vector $\mathbf{w}$)?

(b)

Implement the function

> `lr_train(X, y, eta=0.01, max_iter=2000, alpha=0, epsilon=0.0001, trace=False)`

for training logistic regression models using gradient descent (*batch* approach). The function takes a labeled set of learning examples (the matrix of examples `X` and the vector of labels `y`) and returns a $(n+1)$-dimensional vector of weights of type `ndarray`. If `trace=True`, the function additionally returns a list (or matrix) of weight vectors $\mathbf{w}^0, \mathbf{w}^1, \ldots, \mathbf{w}^k$ generated through all optimization iterations, from 0 to k. The optimization should be carried out until the `max_iter` iteration is reached, or when the difference in cross-entropy error between two iterations falls below the `epsilon` value. The `alpha` parameter represents the L2-regularization factor.

We recommend defining the helper function `lr_h(x,w)` which gives a prediction for the example `x` with the given weights `w`. Also, we recommend the function `cross_entropy_error(X,y,w)` which calculates the cross entropy error of the model on the labeled set `(X,y)` with the same weights.

NB: Please note that the way the labels ($\{+1, -1\}$ or $\{1, 0\}$) are defined is compatible with the calculation of the loss function in the optimization algorithm.

In [360…

```
from numpy import linalg
from math import inf


def lr_train(X, y, eta=0.01, max_iter=2000, alpha=0, epsilon=0.0001, trace=False):
    # Your code here
    w = np.zeros(X.shape[1] + 1)
    X = np.concatenate((np.ones((X.shape[0], 1)), X), axis=1)
    W = np.copy(w).reshape(1, -1)
    err_act = inf

    for i in range(max_iter):
        err_prev = err_act
```

```
        err_act = cross_entropy_error(X,y,w,alpha)
        if(abs(err_prev - err_act) < epsilon):
            print(f'stop because epsilon was reached in iter {i}')
            break
        h = lr_h(X,w)
        w = w * (1 - eta * alpha) - eta * ((h - y).T @ X).T
        if trace:
            W = np.vstack([W, w])


    return w, W




def lr_h(X,w):
    z = X @ w
    return 1 / (1 + np.exp(-z))


def cross_entropy_error(X,y,w, alpha=0):
    N = X.shape[0]
    h = lr_h(X,w)
    return -np.sum(y * np.log(h) + (1 - y) * np.log(1 - h)) + alpha / 2 * w.T @ w
```

(c)

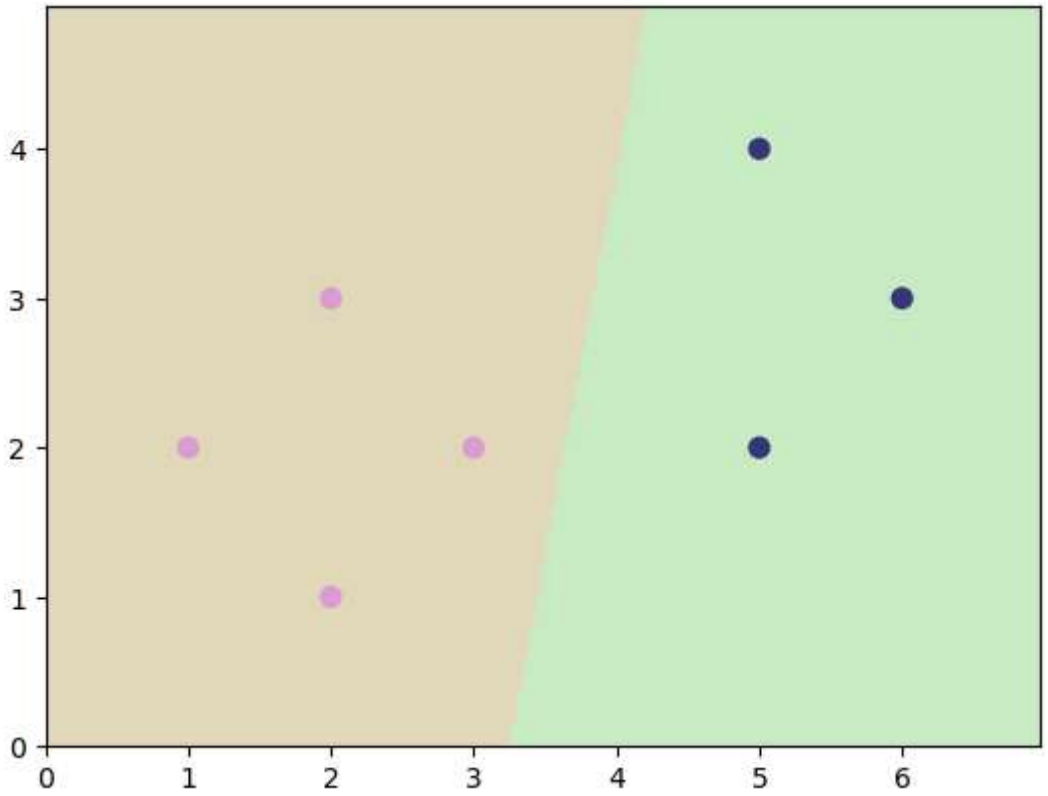
Using the `lr_train` function, train the logistic regression model on the set `seven`, display the resulting boundary between the classes, and calculate the cross-entropy error.

NB: Make sure to pass a sufficient number of iterations to the function.

In [352…
```
seven_X = np.array([[2,1], [2,3], [1,2], [3,2], [5,2], [5,4], [6,3]])
seven_y = np.array([1, 1, 1, 1, 0, 0, 0])
```

In [371…
```
# Your code here
w, W = lr_train(seven_X, seven_y, max_iter=10000)
plot_2d_clf_problem(seven_X, seven_y, lambda x: lr_h(PolynomialFeatures(1).fit_transform(x), w) >= 0.5)
```

```
stop because epsilon was reached in iter 3105
```



Q: Which stopping criteria is activated?

Q: Why is the resulting cross-entropy error not zero?

Q: How would you determine that the optimization procedure has indeed found a hypothesis that minimizes the learning error? What does that depend on?

Q: How would you modify the code if you wanted the optimization to be performed by stochastic gradient descent (*online learning*)?

(d)

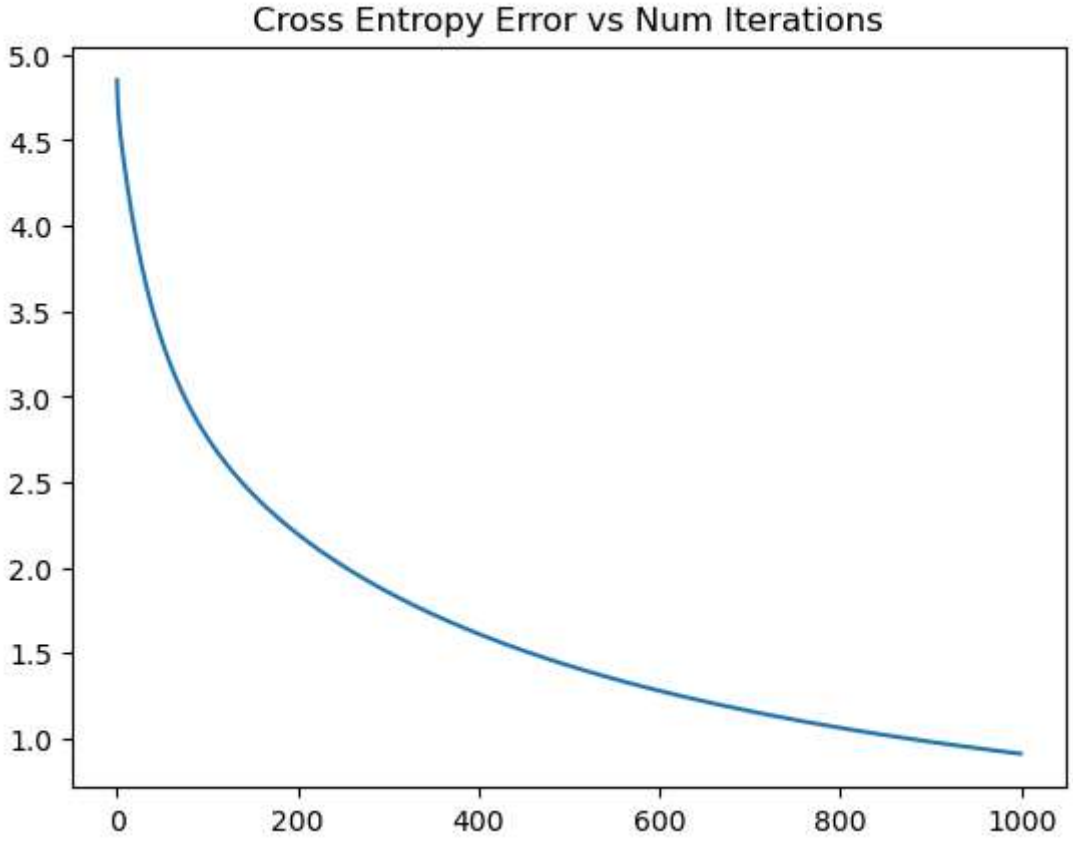
Show in one graph the cross-entropy error (logistic loss expectation) and the classification error (loss expectation 0-1) on set `seven` through iterations of the optimization procedure. Use the trace of the weights of the function `lr_train` from task (b) (option `trace=True`). In the second graph, show the cross-entropy error as a function of the number of iterations for different learning rates, $\eta \in \{0.005, 0.01, 0.05, 0.1\}$.

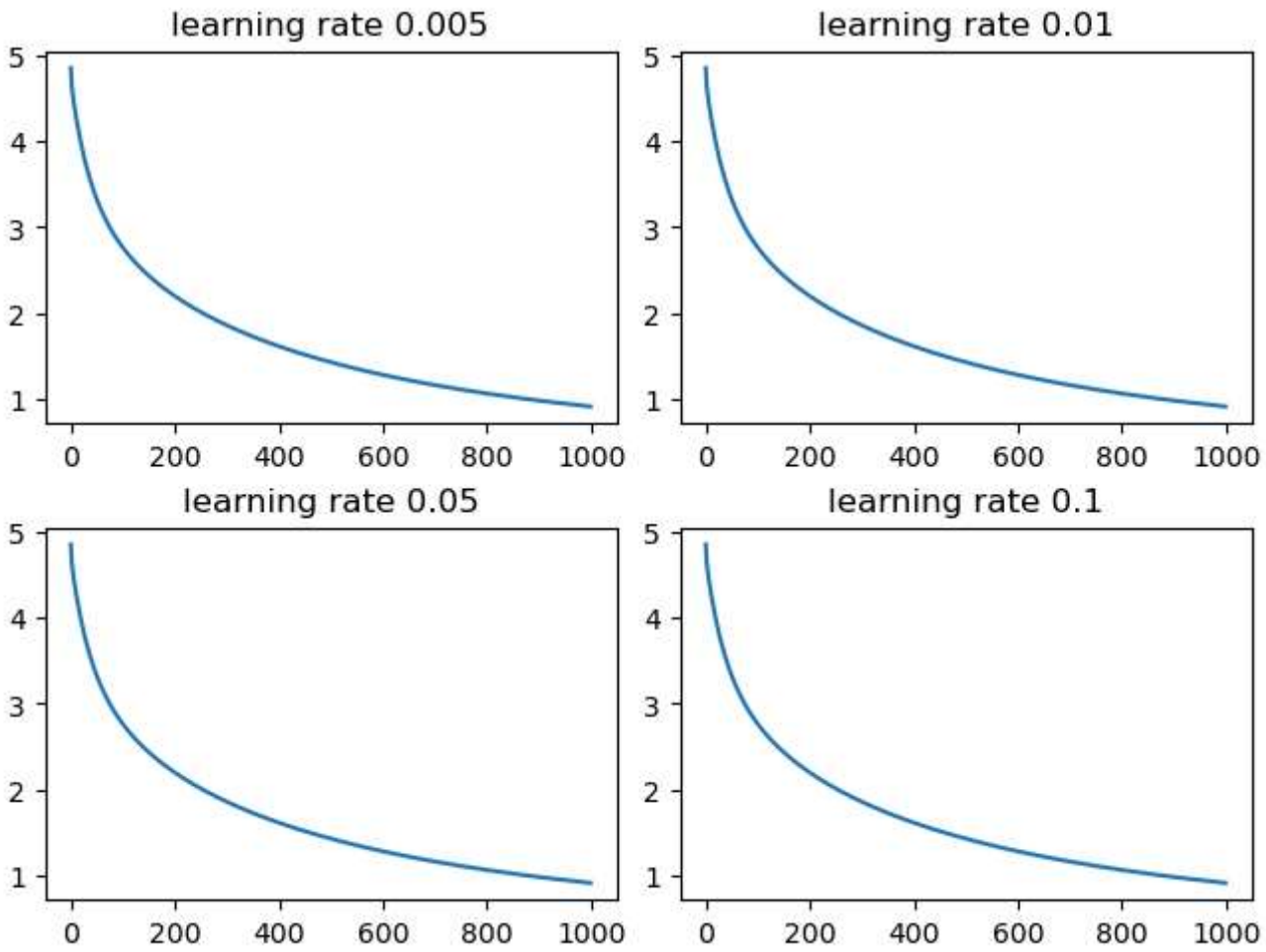
In [311…

```
from sklearn.metrics import zero_one_loss
```

In [391…

```
# Your code here
iter = 1000
w, W = lr_train(seven_X, seven_y, max_iter=iter, trace=True)
plt.figure()
cross_entropy = np.zeros(iter)
for i in range(iter):
    cross_entropy[i] = cross_entropy_error(
        PolynomialFeatures(1).fit_transform(seven_X),
        seven_y,
        W[i])
plt.plot(np.arange(iter), cross_entropy)
plt.title('Cross Entropy Error vs Num Iterations')
plt.show()

plt.subplots(layout='constrained')
plt.axis('off')
learning_rates = [0.005,0.01,0.05,0.1]
j = 1
for lr in learning_rates:
    plt.subplot(220 + j)
    j+=1
    for i in range(iter):
        cross_entropy[i] = cross_entropy_error(
            PolynomialFeatures(1).fit_transform(seven_X),
            seven_y,
            W[i]
        )
    plt.title(f'learning rate {lr}')
    plt.plot(np.arange(iter), cross_entropy)
```


Cross Entropy Error vs Num Iterations
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
200
400
600
800
1000



Q: Why is the cross-entropy error greater than the classification error? Is this always the case with logistic regression and why?

Q: Which learning rate η would you choose and why?

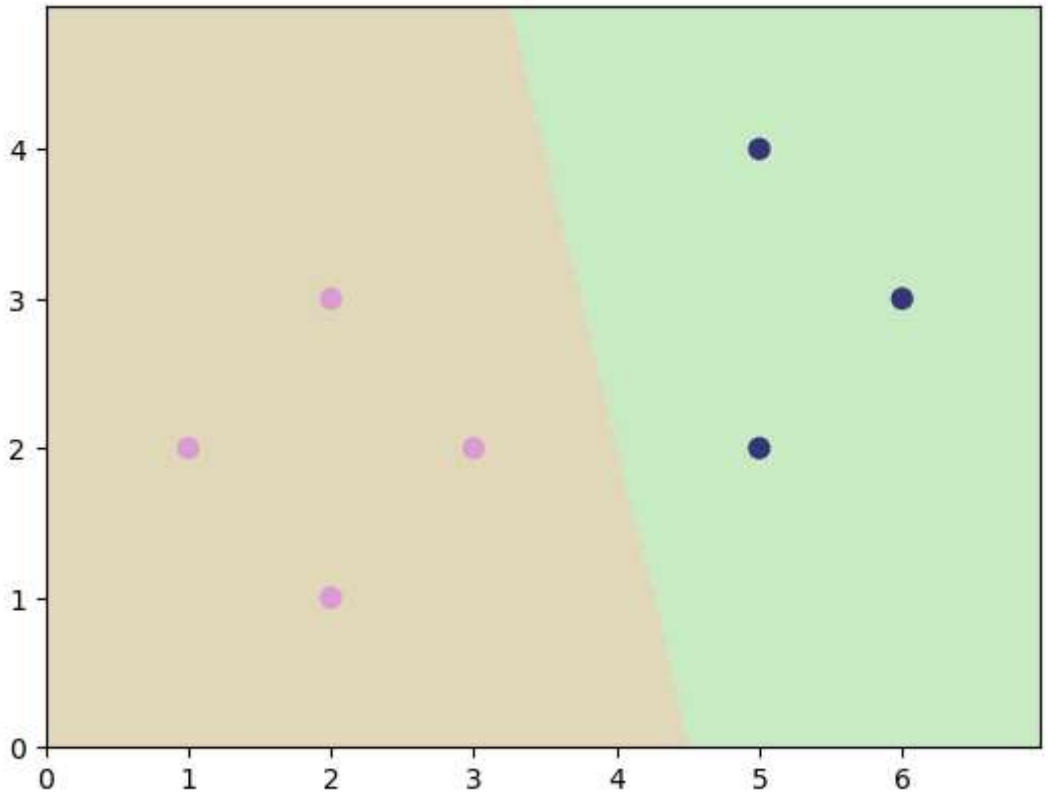
(e)

Familiarize yourself with the `linear_model.LogisticRegression` class that implements logistic regression. Compare the result of that model on set `seven` with the result you get using your own implementation of the algorithm.

NB: As the built-in implementation uses more advanced versions of the optimization function, it is very likely that your solutions will not match, but the general performance of the model should. Again, pay attention to the number of iterations and the strength of the regularization.

In [147…

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression().fit(seven_X, seven_y)
plot_2d_clf_problem(seven_X, seven_y, lambda x : model.predict(x) >= 0.5)
```



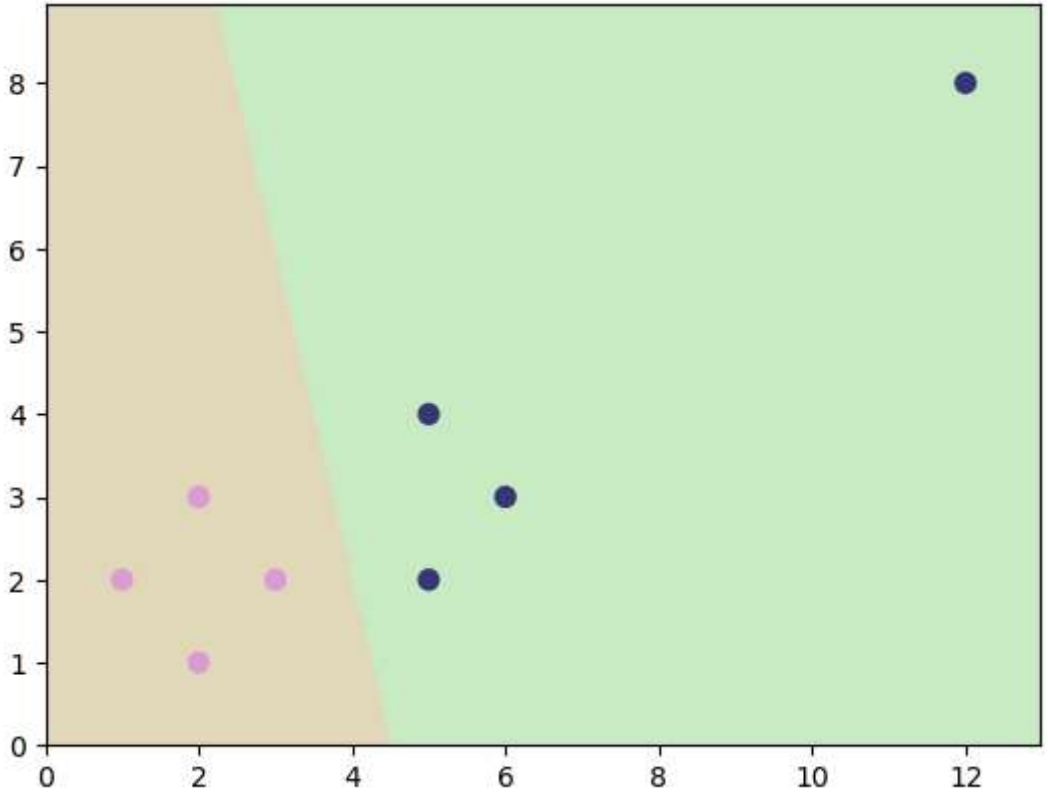
4. Analysis of logistic regression

(a)

Using the built-in logistic regression implementation, analyse how logistic regression handles outliers. Use the `outlier` set. Show the boundary between classes.

In [149…
```
outlier_X = np.append(seven_X, [[12,8]], axis=0)
outlier_y = np.append(seven_y, 0)
```

In [151…
```
# Your code here
model = LogisticRegression().fit(outlier_X, outlier_y)
plot_2d_clf_problem(outlier_X, outlier_y, lambda x : model.predict(x) >= 0.5)
```



Q: Why is the result different from the one obtained by the linear regression classification model from the first task?

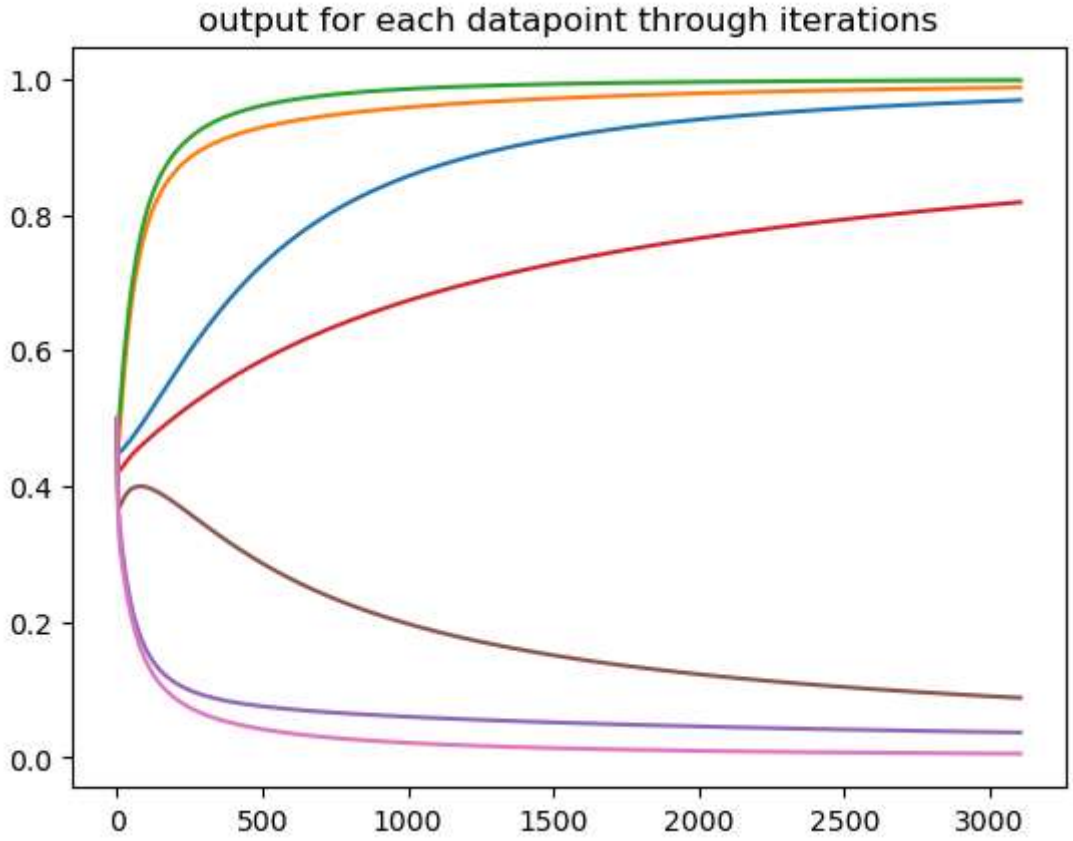
(b)

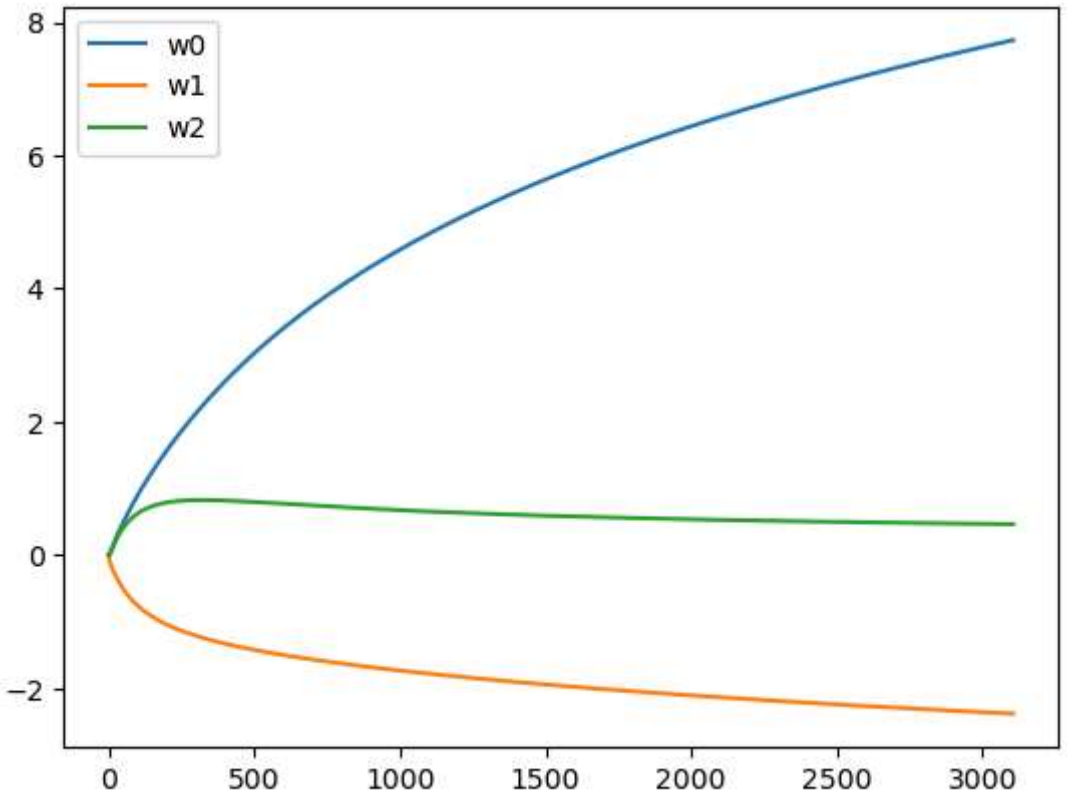
Train the logistic regression model on the set `seven` and show on two separate graphs, through iterations of the optimization algorithm, (1) the output of the $h(\mathbf{x})$ model for all seven examples and (2) the weight values w_0, w_1, w_2.

In [404…

```
# Your code here
w, W = lr_train(seven_X, seven_y, max_iter= 10000, trace=True)

plt.figure()
plt.title('output for each datapoint through iterations')
outputs = np.zeros((W.shape[0], 7))
for i in range(W.shape[0]):
    h = lr_h(PolynomialFeatures(1).fit_transform(seven_X), W[i])
    outputs[i] = h

for i in range(7):
    plt.plot(np.arange(W.shape[0]), outputs[:,i])
plt.plot()

plt.figure()
plt.plot(np.arange(W.shape[0]), W[:,0], label='w0')
plt.plot(np.arange(W.shape[0]), W[:,1], label='w1')
plt.plot(np.arange(W.shape[0]), W[:,2], label='w2')
plt.legend()
plt.show()
```

```
stop because epsilon was reached in iter 3105
```


output for each datapoint through iterations
1.0
0.8
0.6
0.4
0.2
0.0
0
500
1000
1500
2000
2500
3000



(c)

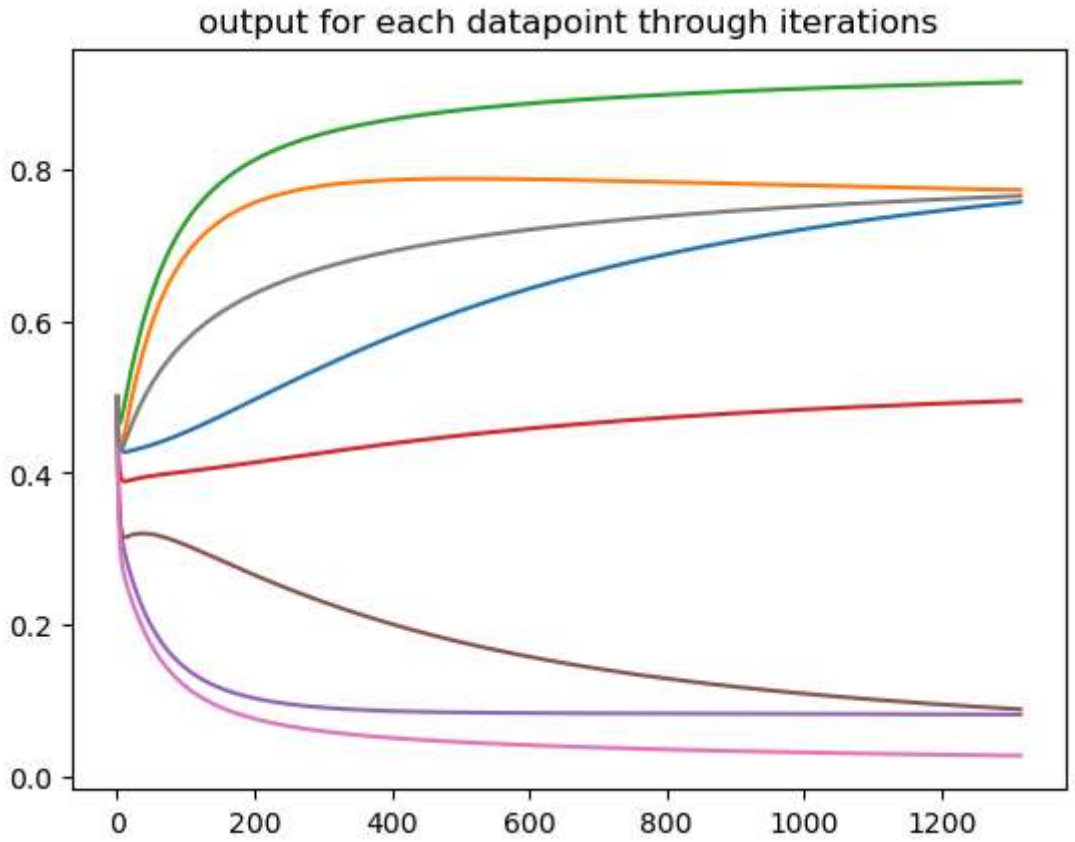
Repeat the experiment from subtask (b) using the linearly inseparable data set `unsep`.

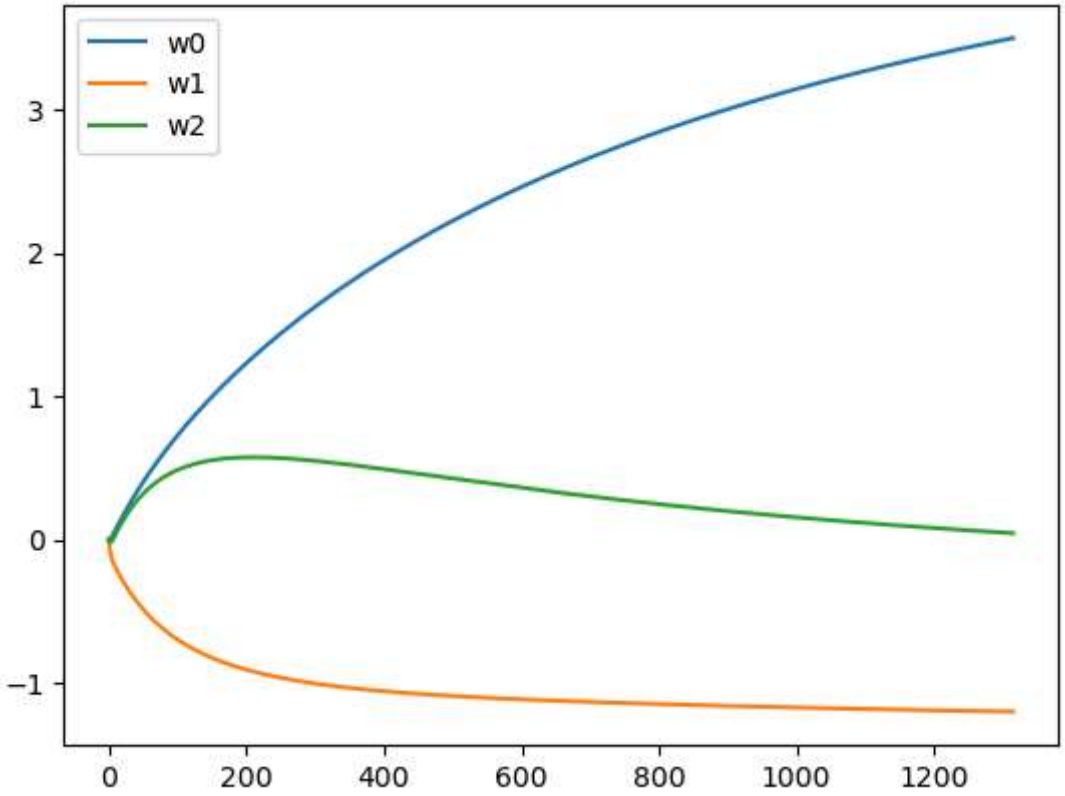
In [406…
```
unsep_X = np.append(seven_X, [[2,2]], axis=0)
unsep_y = np.append(seven_y, 0)
```

In [410…
```
# Your code here
w, W = lr_train(unsep_X, unsep_y, max_iter= 10000, trace=True)

plt.figure()
plt.title('output for each datapoint through iterations')
outputs = np.zeros((W.shape[0], unsep_X.shape[0]))
for i in range(W.shape[0]):
```

```
    h = lr_h(PolynomialFeatures(1).fit_transform(unsep_X), W[i])
    outputs[i] = h

for i in range(unsep_X.shape[0]):
    plt.plot(np.arange(W.shape[0]), outputs[:,i])
plt.plot()

plt.figure()
plt.plot(np.arange(W.shape[0]), W[:,0], label='w0')
plt.plot(np.arange(W.shape[0]), W[:,1], label='w1')
plt.plot(np.arange(W.shape[0]), W[:,2], label='w2')
plt.legend()
plt.show()
```

```
stop because epsilon was reached in iter 1315
```





Q: Compare the graphs for the case of linearly separable and linearly non-separable examples and comment on the difference.

5. Regularized logistic regression

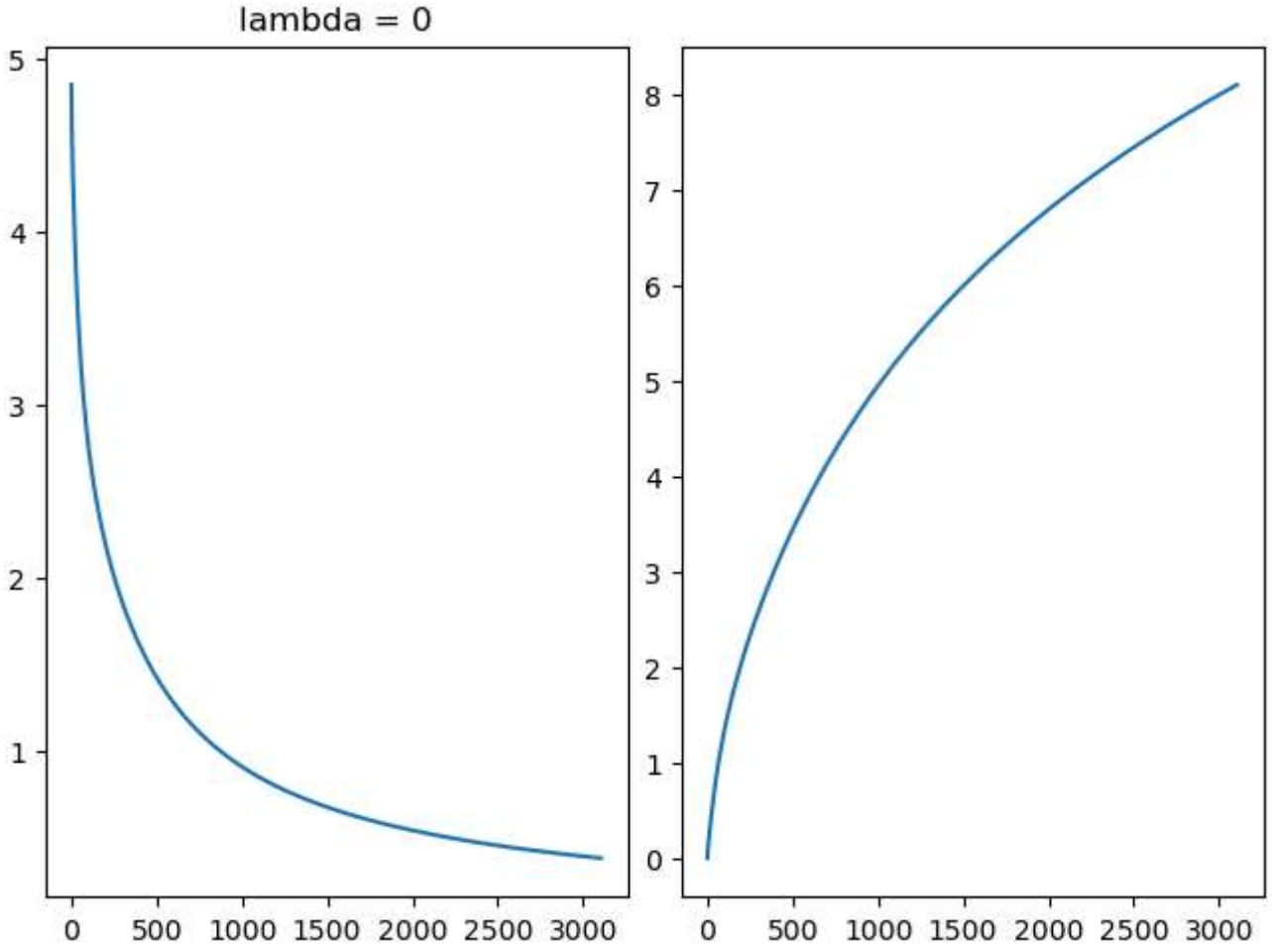
Train the logistic regression model on the `seven` data set with different L2 regularization factors, $\alpha \in \{0, 1, 10, 100\}$. Show on two separate graphs (1) the cross-entropy error and (2) the L2-norm of the vector $\mathbf{w}$ through the iterations of the optimization algorithm.

In [ ]:
```
from numpy.linalg import norm
```

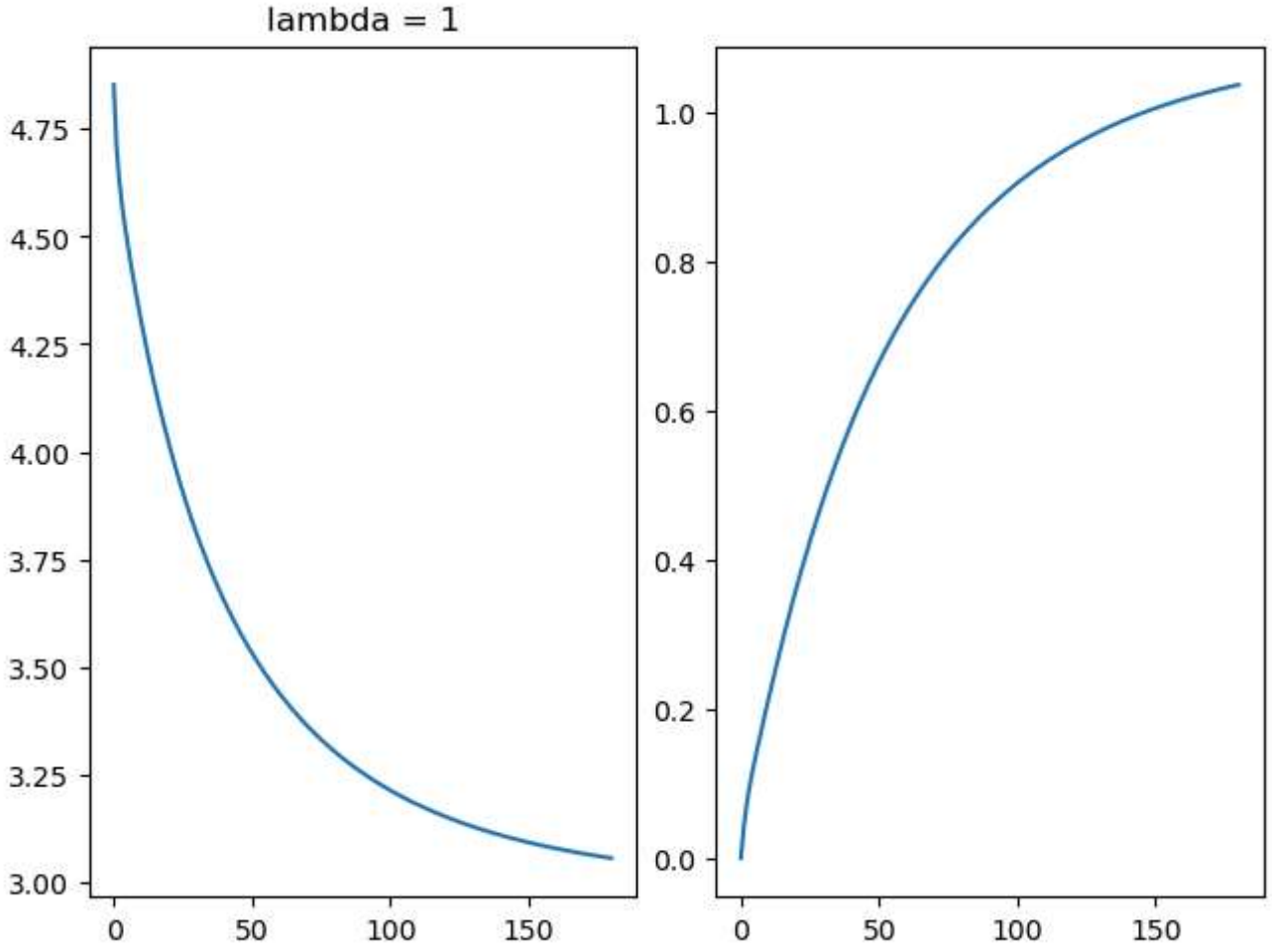
In [432…
```
# Your code here
for reg in [0,1,10,100]:

    plt.subplots(layout='constrained')
```

```
    w, W = lr_train(seven_X, seven_y, max_iter= 10000, trace=True, alpha=reg)
    plt.axis('off')

    plt.subplot(121)
    plt.title(f'lambda = {reg}')
    cross_entropy = np.zeros(W.shape[0])
    norms = np.zeros(W.shape[0])
    for i in range(W.shape[0]):
        cross_entropy[i] = cross_entropy_error(
            PolynomialFeatures(1).fit_transform(seven_X),
            seven_y,
            W[i])
        norms[i] = norm(W[i])
    plt.plot(np.arange(W.shape[0]), cross_entropy)

    plt.subplot(122)
    plt.plot(np.arange(W.shape[0]), norms)

    plt.show()
```

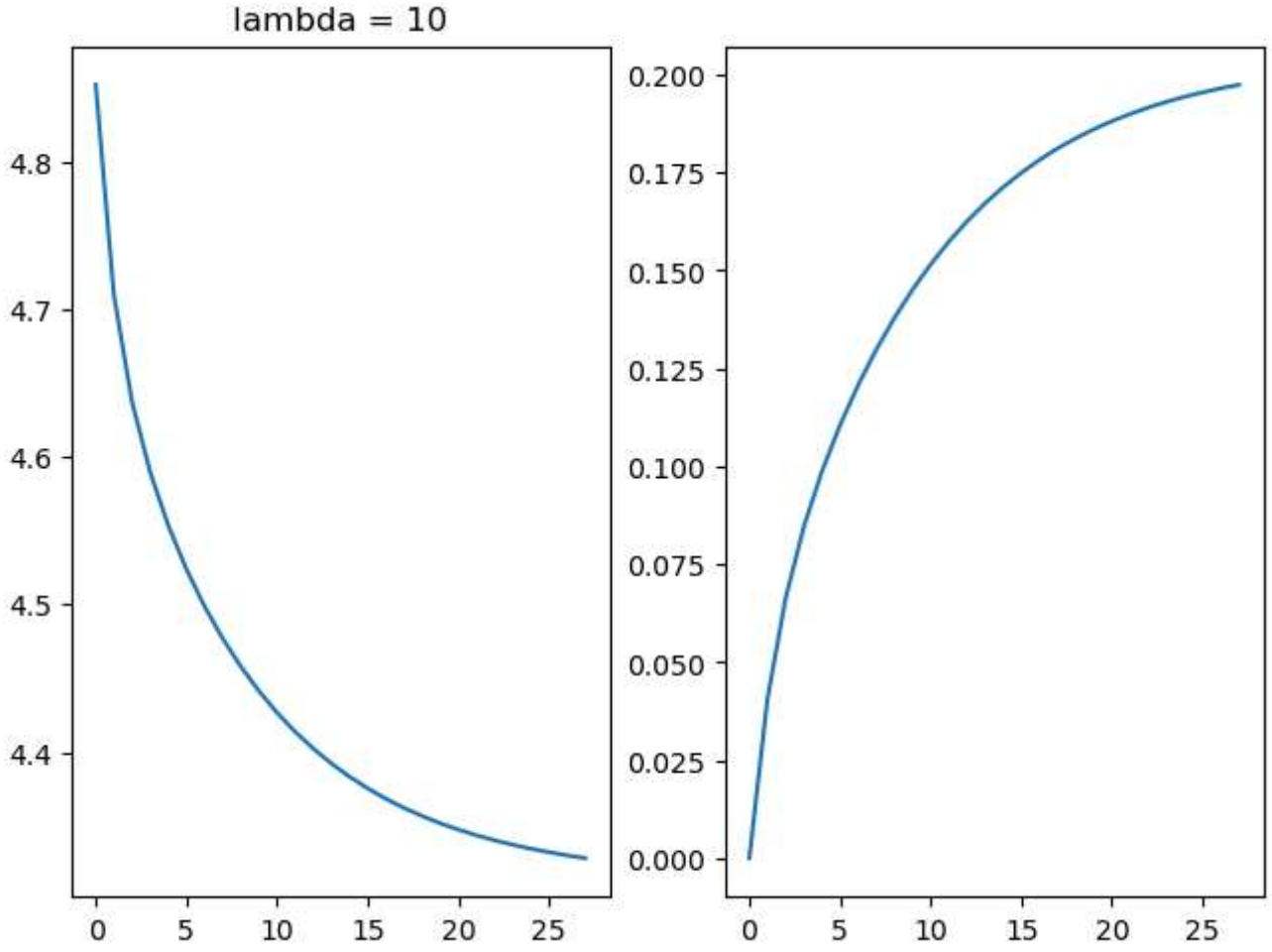
```
stop because epsilon was reached in iter 3105
```



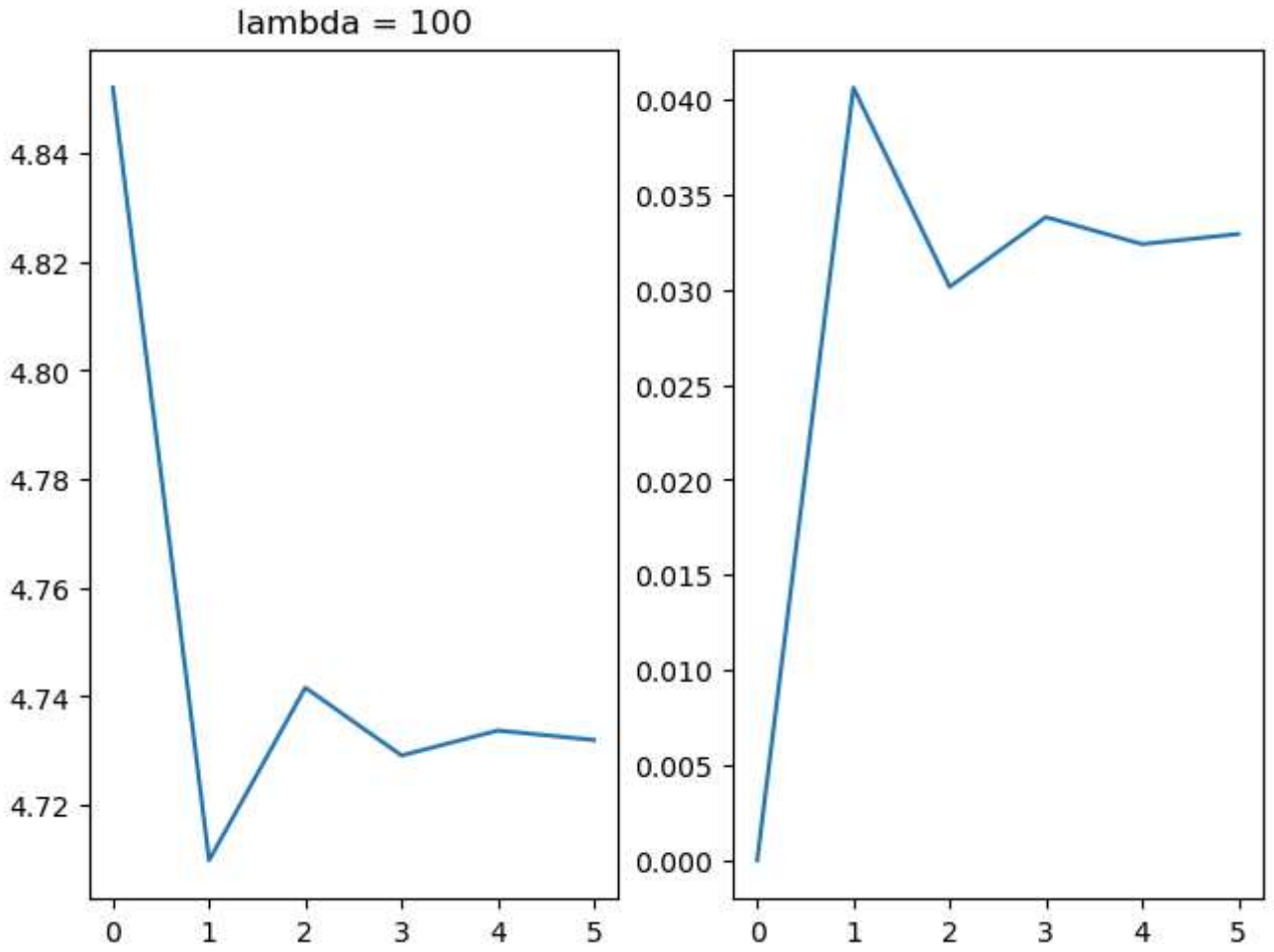
stop because epsilon was reached in iter 180




```
stop because epsilon was reached in iter 27
```




```
stop because epsilon was reached in iter 5
```



Q: Are the curves expected and why?

Q: What value for α would you choose and why?

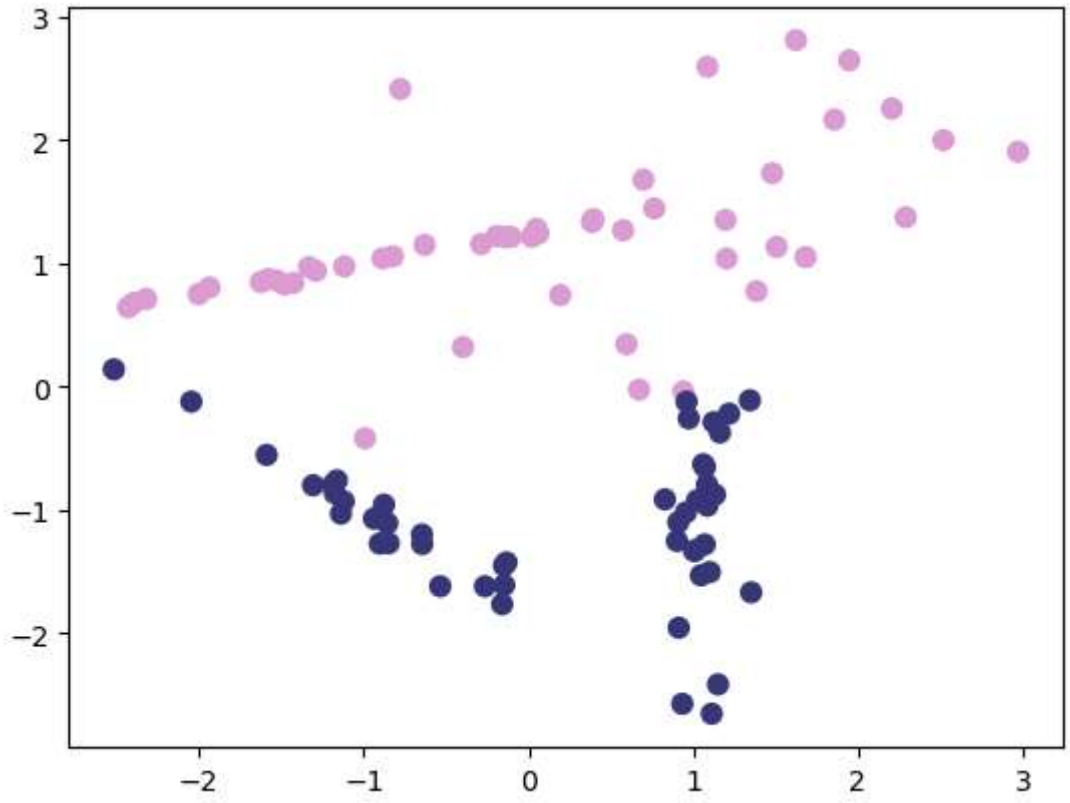
Study the function `datasets.make_classification`. Generate and display a two-class dataset with a total of $N = 100$ two-dimensional ($n = 2$) examples, with two clusters per class (`n_clusters_per_class=2`). It is unlikely that the set generated in this way will be linearly separable, but this is not a problem because we can map the examples to a multidimensional feature space using the class [`preprocessing.PolynomialFeatures`](http://scikit-learn.org/stable/modules/generated/ sklearn.preprocessing.PolynomialFeatures.html), as we did with linear regression in the first lab exercise. Train a logistic regression model using polynomial functions of degree $d = 2$ and

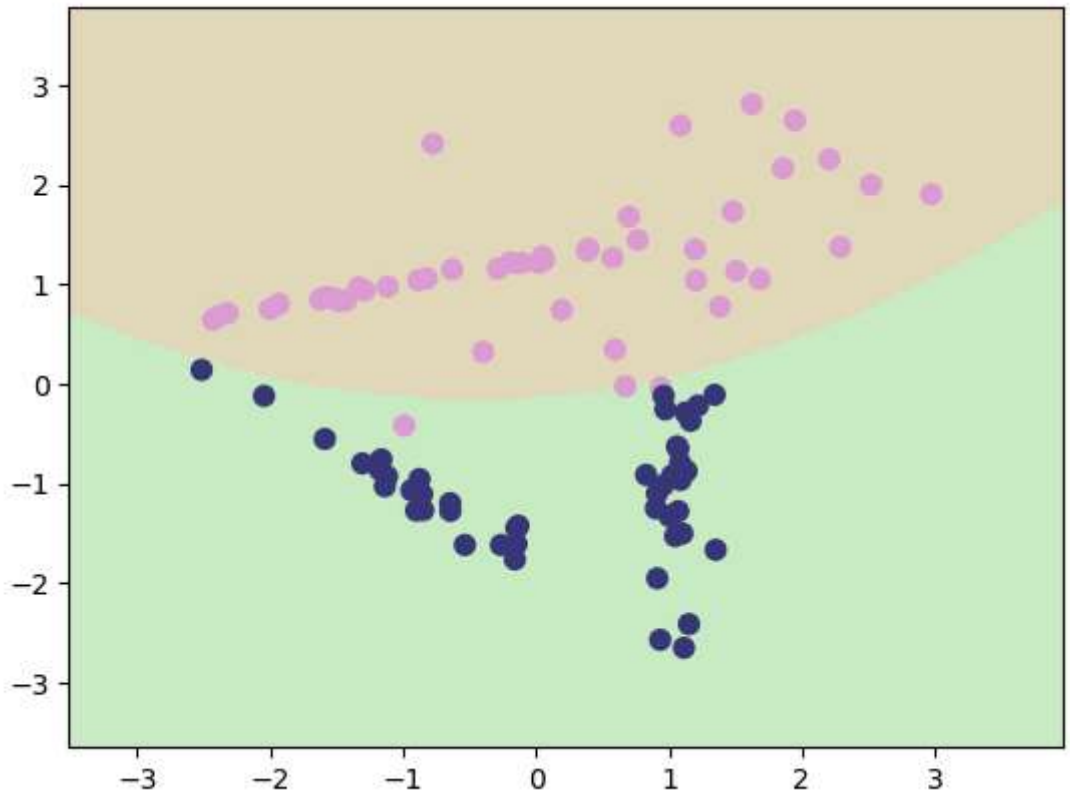
$d = 3$ as mapping functions. Display the obtained boundaries between classes. You can use your own implementation, but for speed it is recommended to use `linear_model.LogisticRegression`. Choose the regularization factor as desired.

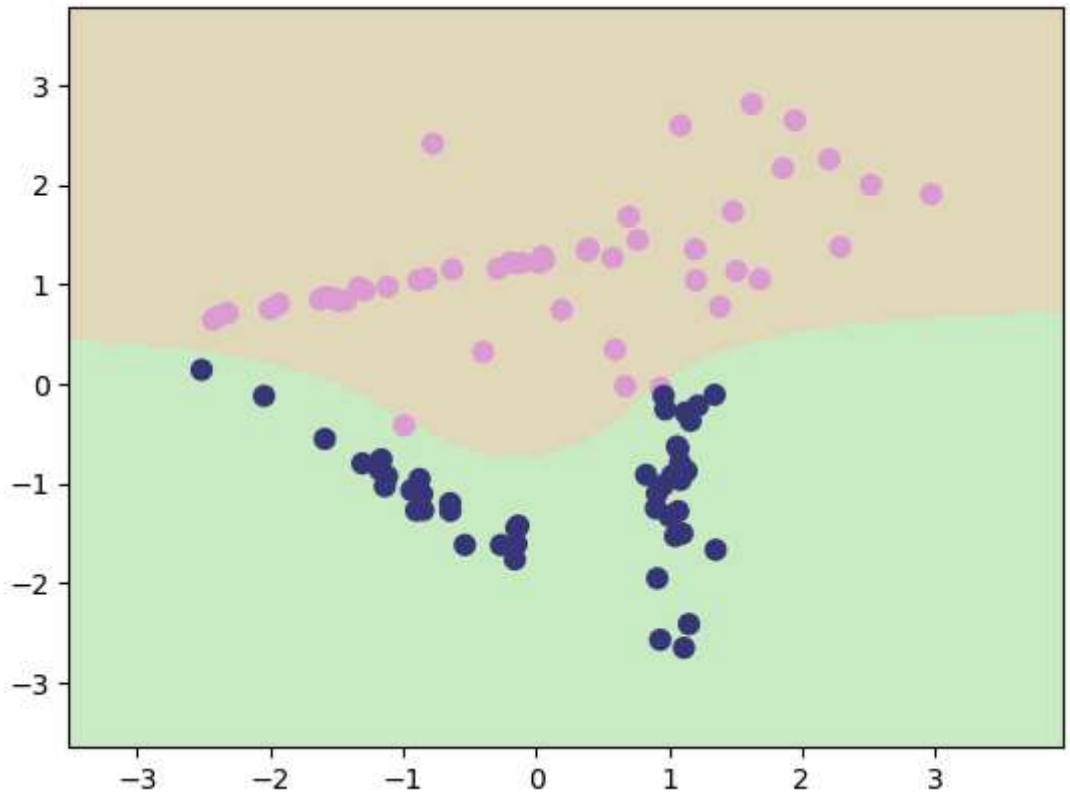
NB: As before, use the function `plot_2d_clf_problem` to display the boundary between classes. Pass the original data set to the function as arguments, and make the mapping into the feature space within the call of the function `h`, which makes the prediction, as follows:

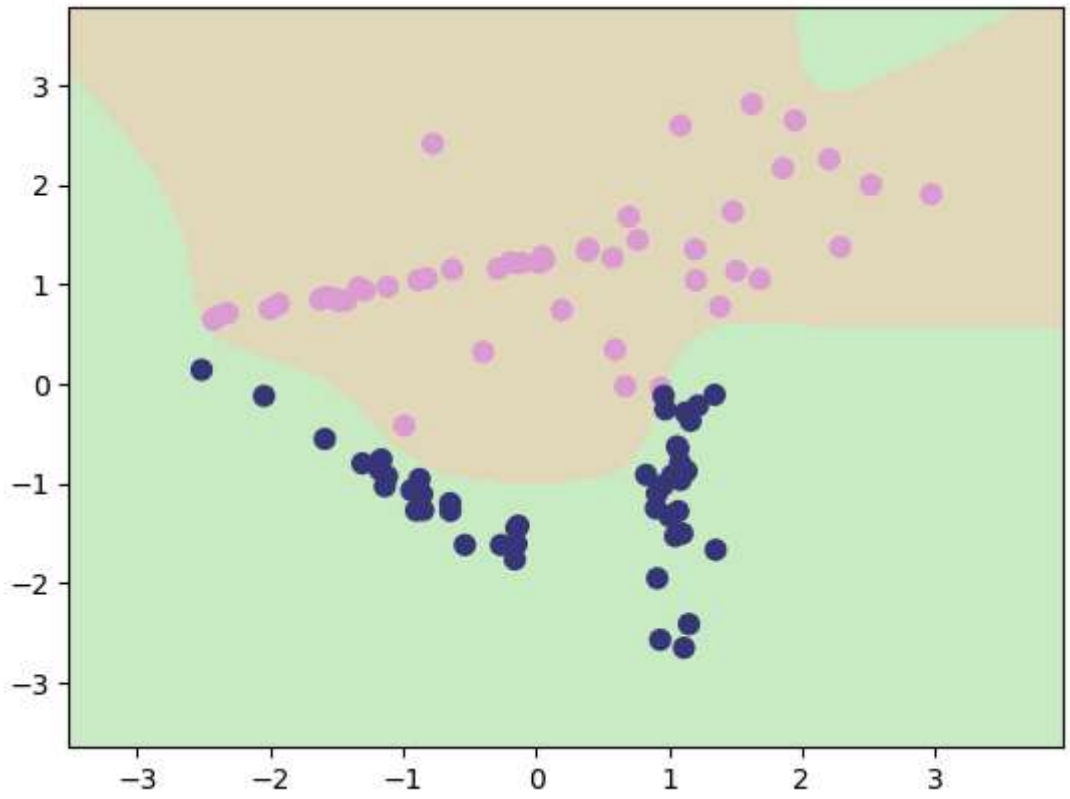
In [454…

```
from sklearn.preprocessing import PolynomialFeatures
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0,
                           n_clusters_per_class=2, n_classes=2)


plot_2d_clf_problem(X, y)
plt.show()

poly = PolynomialFeatures(2)
model = LogisticRegression().fit(poly.fit_transform(X), y)
plot_2d_clf_problem(X,y, lambda x : model.predict(poly.fit_transform(x)) >= 0.5)
plt.show()

plot_2d_clf_problem(X, y)
poly = PolynomialFeatures(3)
model = LogisticRegression().fit(poly.fit_transform(X), y)
plot_2d_clf_problem(X,y, lambda x : model.predict(poly.fit_transform(x)) >= 0.5)
plt.show()


plot_2d_clf_problem(X, y)
poly = PolynomialFeatures(10)
model = LogisticRegression().fit(poly.fit_transform(X), y)
plot_2d_clf_problem(X,y, lambda x : model.predict(poly.fit_transform(x)) >= 0.5)
plt.show()
```


3
2
1
0
−1
−2
−2
−1
0
1
2
3


3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3



Q: What degree of polynomial would you use and why? Is this selection related to the selection of the regularization factor α? Why?